Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 3 of 3 returned.

☐ 1. Document ID: NN9511131

L4: Entry 1 of 3 File: TDBD Nov 1, 1995

TDB-ACC-NO: NN9511131

DISCLOSURE TITLE: Shared Variable Support using Optimistic Execution

SECURITY: Use, copying and distribution of this data is subject to the restictions in the Agreement For IBM TDB Database and Related Computer Databases. Unpublished - all rights reserved under the Copyright Laws of the United States. Contains confidential commercial information of IBM exempt from FOIA disclosure per 5 U.S.C. 552(b)(4) and protected under the Trade Secrets Act, 18 U.S.C. 1905.

COPYRIGHT STATEMENT: The text of this article is Copyrighted (c) IBM Corporation 1995. All rights reserved.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Drawing Desc | Clip Img

☐ 2. Document ID: NB9112357

L4: Entry 2 of 3 File: TDBD Dec 1, 1991

TDB-ACC-NO: NB9112357

DISCLOSURE TITLE: Two Phase Resource Queries with a Suppressible Second Phase.

SECURITY: Use, copying and distribution of this data is subject to the restictions in the Agreement For IBM TDB Database and Related Computer Databases. Unpublished - all rights reserved under the Copyright Laws of the United States. Contains confidential commercial information of IBM exempt from FOIA disclosure per 5 U.S.C. 552(b)(4) and protected under the Trade Secrets Act, 18 U.S.C. 1905.

COPYRIGHT STATEMENT: The text of this article is Copyrighted (c) IBM Corporation 1991. All rights reserved.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Drawing Desc

☐ 3. Document ID: NN79112578

L4: Entry 3 of 3 File: TDBD Nov 1, 1979

TDB-ACC-NO: NN79112578

DISCLOSURE TITLE: Debugging System Compatible With Optimizing Compiler. November 1979.

SECURITY: Use, copying and distribution of this data is subject to the restictions in the Agreement For IBM TDB Database and Related Computer Databases. Unpublished - all rights reserved under the Copyright Laws of the United States. Contains confidential commercial information of IBM exempt from FOIA disclosure per 5 U.S.C. 552(b)(4) and protected under the Trade Secrets Act, 18 U.S.C. 1905.

COPYRIGHT STATEMENT: The text of this article is Copyrighted (c) IBM Corporation 1979. All rights reserved.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Terms	Documents
L3 and presence	3

Display Format: TI Change Format

Previous Page Next Page Go to Doc#

L4: Entry 1 of 3 File: TDBD Nov 1, 1995

TDB-ACC-NO: NN9511131

DISCLOSURE TITLE: Shared Variable Support using Optimistic Execution

PUBLICATION-DATA:
IBM Technical Disclosure Bulletin, November 1995, US

VOLUME NUMBER: 38
ISSUE NUMBER: 11
PAGE NUMBER: 131 - 136

PUBLICATION-DATE: November 1, 1995 (19951101)

CROSS REFERENCE: 0018-8689-38-11-131

DISCLOSURE TEXT:

This document contains drawings, formulas, and/or symbols that will not appear on line. Request hardcopy from ITIRC for complete article. A method for virtual time execution with variables shared between two or more communicating sequential processes is disclosed. The technique disclosed here clusters and encapsulates shared variables within an implicit process. Many languages used for hardware description and general purpose programming share variables between two or more processes. Examples include Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) (1), Verilog (2), and concurrent dialects of C++ (3). The presence of shared variables in programs presents a challenge to efficient parallel execution, especially if parallel execution is to achieve results equivalent to a serial execution of the same program. A means has been disclosed (4) of executing Communicating Sequential Process (CSP) programs which helps to decouple execution of processes, and thus would tend to improve parallel execution. His technique is variously referred to as Optimistic, Virtual Time or Time Warp execution. This disclosure will subsequently refer to the technique as optimistic execution.

In order to preserve the semantics of a sequential execution during a parallel execution, optimistic execution assumes there is a global logical time domain in which execution ordering constraints can be defined. Each process has a Local Time (LT), expressed in the logical time domain, and used to order evaluation within the process and communication between processes. A Global Virtual Time (GVT) denotes the earliest local time of any process. A distinct real time domain, sometimes known as wall-clock time, is used to refer to the order in which hardware processors actual execute instructions.

Under optimistic execution, each process progressively processes the messages at later points in logical time assuming that it will not receive any message with the logical time less than the logical time of these messages. Such an execution continues until the process is notified that the execution was based on an invalid assumption. Notification of an invalid assumption takes the form of a message arriving at the process either with the logical time less than the logical time of the executed messages(s) or a message indicating that the executed message at a particular logical time should be cancelled.

In response to notification of an invalid assumption, the process must be prepared to rollback both its state and all of the messages sent by the process, to the state present before the (invalid) assumption was made. Message rollback occurs by sending an anti-message for each message which must be rolled back. Conceptually messages and anti-messages combine to eliminate each other. In order to minimize the overhead associated with executing processes using optimistic execution, (physical) processes created during source code

analysis or static elaboration are often clustered to form logical processes. Clustering is a common step for practical use of optimistic execution; its requirements are generally language specific. The following assumes that clustering has occurred to form logical processes. These logical processes will simply be referred to as processes, Pn (for various n), below. In a like manner, shared variables may optionally be clustered into subsets to improve optimistic execution performance. Each subset is disjoint from any other subset. The union of all subsets constitutes the set of all shared variables referenced from at least one process. If at least one shared variable is referenced by two processes, Pn and Pm (for any n not equal to m), then all shared variables common to Pn and Pm may belong to the same subset. This clustering is conceptually repeated for all pairings of Pn and Pm.

During elaboration, each shared variable subset defined above is encapsulated by a (new) shared variable process, SVPn. Such shared variable processes provide for atomic operations on the encapsulated shared variables and may provide for language-specific mutual exclusion around dynamically allocated storage accessible from one or more processes. Statements within processes may reference shared variables using a variety of atomic operators. These operators range from simple lock, read, write and unlock to arbitrarily complex atomic operations such as insert database record into a database structure. Some form of lock and unlock are requisite for support of most languages with shared variables. Some languages provide support for arbitrary atomic operations on shared variables, such as (5) protected type mechanism. Each atomic operation executed by a statement in a process is logically implemented as a message sent to an SVP and at least one response from the SVP (see discussion of antimessages below). Even with optimization, such transactions have substantial overhead. In order to reduce the impact of message overhead, abstract atomic operations are preferable over simple atomic operators. Depending on language semantics, each shared variable may have its own (implicit) lock within the SVP or multiple shared variables in the same SVP may share a common lock. Some language semantics may even associate several locks with the same shared variable in order to implement more complex mutual exclusion semantics. Ultimately each abstract operator defined for the same shared variable might even have its own lock. If several shared variables within an SVP can retain access or pointer values of the same type (or convertible type), they might also share a common lock. All such locks are local to a single SVP and implicitly implement a language's mutual exclusion semantics. When a process sends a message initiating an atomic operation on a shared variable encapsulated in an SVP, the sending process must appear to spin-lock awaiting a response. If the message send is explicit (not optimized away), then the sending process may actually suspend execution until a response arrives back at the process from the SVP. When the SVP receives a message, the SVP may either respond to the request immediately (if the required lock(s) are free) or may queue the request pending lock availability. Each time a lock becomes free (as the result of an explicit or implied unlock), the SVP re-evaluates any applicable request queue pending lock availability.

If a SVP receives two messages with the same logical time, one message will receive a response before the other, however languages seldom prescribe what order simultaneous requests will be handled in. However, languages often place constraints on such simultaneous requests coming from distinct processes, such as declaring that two atomic write requests at the same time or both read and write request operations must be detected as an error. Implicit state and instructions executing inside the SVP may enforce these kind of language-specific restrictions. In the absence of any ordering restrictions on simultaneous requests, multiple executions of the program or model with the same input may result in multiple, distinct but correct outputs.

Fig. 1 illustrates an example where A is a shared variable which is written by one process and read by another. Fig. 2 shows the sequence of messages and how the SVP records the execution history. Each message in the figure takes the form of a tuple. The tuple includes the message operation field (e.g., lock, read, write, unlock, insert ...), the logical time (or ordering) associated with the message (e.g., 1, ST2 ...) and the real time at which the SVP receives the message (e.g., RT1, RT2, ...). Fig. 2 shows the order in which the messages are sent, ordered in real time. In the example shown in Fig. 1, the logical message ordering and the order in the real time domain are the same, thus no assumptions are invalidated and no rollback need occur. However, in general messages may be received out of order as observed by the receiving process (or SVP). Fig. 3 illustrates a situation where messages arrive out of order, resulting in rollback. At real time RT1, a process P1 writes a value '0' with a local time of 0 within the SVP. For the same shared variable, at real time T2 >

T1, process P2 writes a value '1' at it local simulation time 100. The value written by P2 is read by process P3 at real time T3 > T2 when its local simulation time is 1000. Given the scenario established by Fig. 3, let us say that process P1 wants to write a value '0' for the shared variable at real time T4 > T3 and P3's local simulation time is 500. In this case the value read by P3 at real time T3 is not the correct value. Hence, we would need to rollback all the simulation activities done by P3 at local simulation time 1000. The required rollback occurs by sending an antimessage from the SVP to P3 in response to its read request. In order to satisfy the canceled response from P3 for the value of the shared variable at 1000, the SVP sends the new updated value of the shared variable to P3. Upon receiving the antimessage, P3 will rollback all the simulation activities done at simulation time 1000 and will use the new value of the shared variable to advance P3's local time. This disclosure shows a means of supporting shared variables within an optimistic execution environment. The technique is suitable for a wide variety of shared variable semantics and (orthogonal) optimistic execution optimizations. References (1) "IEEE Standard VHDL Language Reference Manual," ANSI/IEEE Std. 1076-1993, IEEE Press, (1994). (2) "IEEE Draft Standard VHDL Language Reference Manual," ANSI/IEEE Std. 1364-1995, IEEE Press, (1994). (3) Bjarne Stroustrup, "The C++ Programming Language," Addison Wesley, (1991). (4) D. R. Jefferson, "Virtual Time," ACM Transactions on Programming Languages and Systems, Volume 7, Number 3 (July, 1985).

(5) "ANSI Standard Ada Language Reference Manual 1995" American National Standard Institute (1995).